Multithreading and Reactive Programming

19

LEARNING OBJECTIVES

After completing this chapter, you will be able to understand:

- Reactive programming.
- Multithreading and how to program.
- Concurrency API improvements.
- How to deal with individual threads.
- Synchronization blocks.
- Deadlocks and how to resolve deadlocks.
- Concurrent data structures.
- How to design concurrent Java programs.
- Controlling sequential executions.
- How to avoid lazy initialization.

19.1 | Introduction

Java 11 is the latest version of Java released by Oracle, which has some interesting features. Before its release, Java 9 brought major updates. Java 9 updates offer improved support to large heaps as there is great demand for improving the capacity of Java to handle large memory sizes that are required to support cloud applications. It follows a module system and includes several Java projects that were initially defined for the new release.

Our focus in this chapter remains on discussing the improved multithreading capabilities of Java. We will begin this chapter by describing concepts such as reactive programming and multithreading, especially with their application using the new and improved Java's concurrency utilities. We will also cover thread transition, thread interaction, and `newworkstealingPool()` method. With the exercises and other tools provided in this chapter, we believe that Java programmers will learn to use multiple threads in their programming and ensure that their programs are efficient by employing the available capacities in modern processors.

19.2 | Reactive Programming

Reactive programming is a special method that employs an asynchronous style. It refers to a method that employs improved control over data streams and data streams use in changing the way the programming behaves with the future data stream. All kinds of data streams can be expressed using reactive methods and the programs can be designed to execute different changes automatically, according to the data flow that they receive from program outputs and other important parameters.

Reactive programming works well when employed in an imperative setting. The use of relations and the effective change in program behavior is high possibility in Java. In fact, any language that can directly control and describe hardware components such as Verilog can benefit from the use of reactive programming. It gives improved control over the available hardware resources.

Reactive programming works well by understanding that there is a publisher that keeps producing data, while there is also a subscriber that requires asynchronous access to the process information. This relationship is best described by Figure 19.1.

Figure 19.1 Publisher and subscriber relationship for reactive programming.

There are several benefits to implementing reactive programming. Here are two common advantages that you get as a reactive programming expert:

1. You can use a simpler code for the required tasks. This allows you to create a readable code, which is easy to implement as well as improve when required in a client application.
2. It allows programmers to focus on implementing business logic in their programming solutions. This takes them away from following a conventional boilerplate code to stay away from similarity and come up with unique programming solutions.

There are four key attributes of a reactive system, which are described in the reactive manifesto. This is illustrated in Figure 19.2.

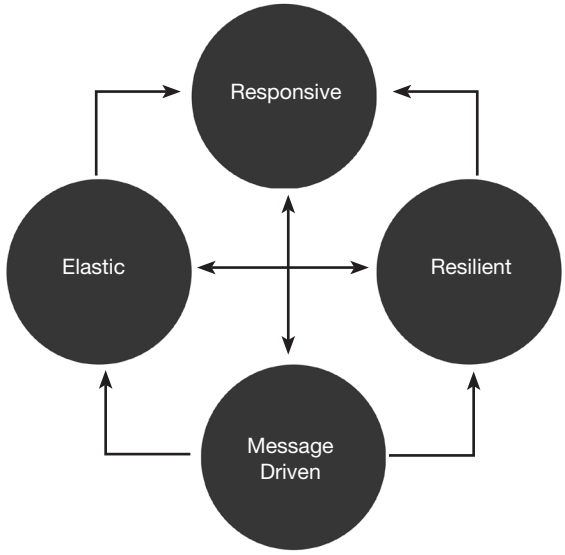

Figure 19.2 Four key attributes of reactive systems.

1. **Responsive:** The system should respond in a timely manner, it not only ensures usability but also effectiveness.
2. **Resilient:** The system should respond in case of failure as well. This can be achieved by focusing on containment, replication, delegation, and isolation.
3. **Elastic:** The system should stay in elastic state. In other words, it should respond to varied workload. In the case of high workload, resource allocation should increase. In case of low workload, resources can be released.
4. **Message driven:** The system should ensure loose coupling, isolation, and location transparency by establishing boundaries between components and relying on asynchronous message passing.

Flash ? Quiz What is the use of "reactivity" in Java?

Concurrency issues are better controlled with reactive programming techniques. The need for creating low-level threads that often depend on ideal synchronization is significantly reduced, thereby resulting in the creation of an environment which is conducive to efficient programming.

Reactive programming improves the efficiency associated with memory use. It is possible to create different streams that are then implemented with a multithreading mechanism to further enhance the results of employing reactive programming through the ideal APIs.

Reactive programming is a successful model with the ability to offer great results in all kinds of programming problems. It is not limited to only offer improvements in real-time applications, as it can be implemented to create reactive hardware definitions that use multithreading schemes.

This model is great for introducing powerful functions that use the available processors to carry out data transformations. There is no need to change either the subscriber or the publisher in the programming model. There can an *N* number of processors that can work on incoming data streams and then provide results to the subscriber according to the particular needs of the application. This technique ensures that programming nodes can be made independent and have the capacity to handle any situation by simply changing the rules that work on the received data streams issued by the publisher.

Java 11 is specially designed to facilitate cloud applications that run in a real-time environment. This means that the language must provide resources that allow for a speedy process and ensure that it is possible to provide the best output, according to the dynamic inputs controlling the situation.

Reactive programs are interactive and are ideal to deal with the changing environment, which is the need in a data center providing support to cloud applications. However, reactive programming through Java interface also has several other applications. It certainly has the ability to create hardware drivers, provide a better virtual machine, and improve protocols that handle data streams.

How will you lose stream data in case of accessing it via concurrent program?

19.3 | Reactive Programming

Since Java 9 update, Java has great potential to carry out reactive programming. In fact, the reactive style is the best one that you can use in Java as it works well with conditions where you can declare the direction that the program must take in the case of receiving specific inputs and processing requests. Since Java 9 update, Java has Flow API that allows the use of reactive streams. This allows programmers to create and filter observable objects that can then be used to implement a dynamic behavior change, whenever it is required in any setting.

Spring is a popular Java framework in this regard, which can be employed for implementing strong reactive patterns and creating applications with the ability to show resilient behavior and improved performance. Throughout the course of this chapter, we will describe various resources and show relevant examples that will allow you to use several APIs to implement reactive programming that performs well in dynamic needs. These are the needs that are specifically required when working with cloud-based applications that need to use large heap sizes and perform better when there are increased processing needs.

The Flow APIs in JDK 9 and after are now working according to the Reactive Streams Specification. There are various implementations which support this standard, while the main goal remains the application of the reactive programming framework that was described earlier.

The API uses a model which depends on a push and a pull model. The Observer is a push mode, where source data items are pushed to reach the application. One the other hand, the pull comes from the Iterator in the Flow API, where the application actively pulls items present at the data source. The API runs by first requesting *N* data items and the publisher then pushes a maximum of these *N* items to the subscriber. The forwarded items may be less if required. The Flow API therefore performs by carrying out a mixture of pull and push steps for reactive functionality. Here is an example of how this API may function, which was taken from the Oracle Community Site (https://community.oracle.com/docs/DOC-1006738):

```
public static interface Flow.Publisher<T> {
    public void subscribe(Flow.Subscriber<? super T> subscriber);
}
public static interface Flow.Subscriber<T> {
    public void onSubscribe(Flow.Subscription subscription);
    public void onNext(T item) ;
    public void onError(Throwable throwable) ;
    public void onComplete() ;
}
public static interface Flow.Subscription {
    public void request(long n);
    public void cancel() ;
}
public static interface Flow.Processor<T,R> extends Flow.Subscriber<T>,
Flow.Publisher<R> {
}
```

This describes the flow process of a typical Java API that employs reactive programming. It uses the pull and push phases that were discussed. We will further present how to employ the individual functionalities from the subscriber and the publisher that are present at the operating end of a Java code that uses reactive programming streams and principles.

QUICK CHALLENGE Take the example of a bank ATM and write a program using the reactive programming concept which allows user to withdraw cash from any ATM.

19.4 | What is Multithreading?

In the context of computer processing, multithreading refers to the capability of using multiple execution threads that can occur independently of each other through a unified process. The threads use the same pool of resources but have different bits of information that require processing (including exception handles), the CPU register requirements, and the stacking status in the addressing space.

The use of multiple threads is an excellent approach for empowering processes on a single processor system. This allows the use of at least two threads where one can always be responding to the user, while another one may be in working condition. However, since most modern computers have multiple processors, the power to employ multiple threads creates an ideal concurrency solution.

Multithreading is also associated with the need to program in a careful manner. It can cause deadlock and conditions where the processor finds it difficult to make the ideal execution decisions. 64-bit operating systems such as Windows often employ pre-emptive multithreading where the software is responsible for switching between different threads. This allows for switching to a high priority thread while using a low priority thread as the trigger element. Another method is the use of cooperative multithreading, which is extremely powerful but creates deadlocks if there is any problem during the execution of a single thread.

Most multithreading occurs at a blistering pace, in which the available time slices into many pieces in the fraction of a second and queued for different threads. This gives the impression to the user that all the threads are running in parallel, when in actual reality, they are taking turns during the processing cycles which is too fast for a typical application.

The primary advantage of using this method is to efficiently employ the available computational resources. A single thread may not often employ the available cache in a physical system. With the presence of multiple threads, it is possible to use the available CPU resources more efficiently, because most of them depend on the results of the current execution.

Running different threads ensures that the resources do not remain idle and will be employed by one or the other thread from time to time, within the microsecond execution cycles. However, multiple threads may interfere with each other if their use is not carefully coordinated. They also face problems if used at lower frequencies. Modern computers have significantly reduced these problems and provide an ideal scenario in which multithreading can be employed with improved accuracy and less failures.

Hardware manufacturers such as Intel claim that the use of multithreading can cause a 30% improvement in the performance of their processors and related components. Processes that often require floating point operations may gain as much as double the performance with parallel processing.

This means that with the right exposure allowed to software elements, multithreading can significantly reduce the execution time of a computer program, allowing for swift processing. It improves user experience and provides important processing advantages with reduced stop-times during executions.

19.4.1 Multithreading in Java

Java is often employed to create programs that may serve several users through a single supporting platform. The thread in Java can be best defined as the smallest processing unit which can be employed to implement multitasking in a program environment. All threads share common heap, thereby ensuring efficient use of the available memory. The switching occurs according to a well-defined context which always happens faster than the time required for the processing of a thread.

Multithreading in Java is an excellent way of improving the performance of programs that produce animations and complex decisions such as games. It is an excellent technique because it can produce the simultaneous results required in such programming environments. You get the following advantages when using multithreading in Java:

1. It is possible to run multiple operations in your program at once, thus reducing the overall time required to produce a response for the program user.
2. Threads are independent in terms of their program decision-making, which allows the user to always have an available interface.
3. Independent threads are excellent for handling program execution exceptions. Stopping one thread does not stop the entire execution and simply alters the resource allocation to produce excellent user interaction.

Understanding the life cycle of a Java thread is important. A thread goes through four separate phases (i.e., if we do not count the running phase), which actually does not affect the thread itself. The four phases are:

1. New phase.
2. Runnable phase.
3. Non-runnable phase.
4. Terminated phase.

Figure 19.3 is a visual presentation of how these phases interact with each other, including the actual running of the thread.

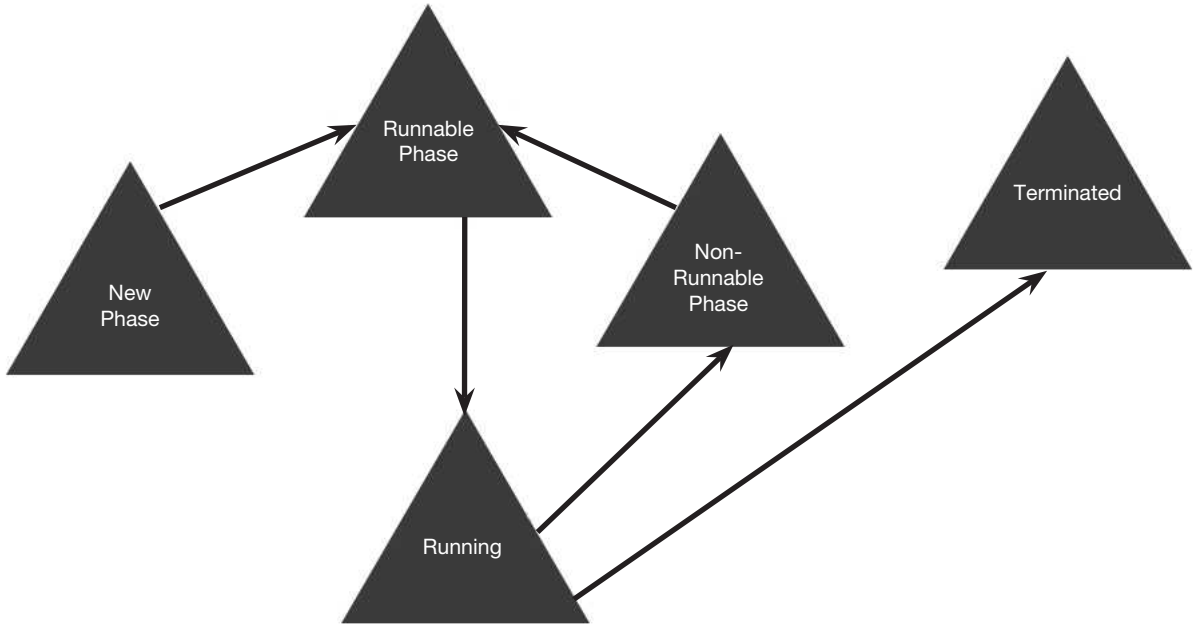

Figure 19.3 Thread life cycle.

The new phase is the initial phase of any thread, which is created when the Java program calls for a Thread class instance. However, this is the phase of the project before the invoking of the `start()` method that produces the next phase.

With the above method employed in the code, the thread is now ready for execution. This is termed as *runnable phase*. However, the actual running of the program is based on when the thread scheduler places the thread on a processing schedule.

The thread that enters the *non-runnable phase*, where it is blocked from further additions. This is important because the thread is still alive and any modification or further processing can cause program errors. The termination is identified when the `run()` method exists for the thread. This is also termed as *dead state*.

QUICK CHALLENGE Explain the thread life cycle concept using a real-life example of a bank ATM in which the user can withdraw cash from any ATM.

See Figure 19.4 to understand the thread states from a different view.

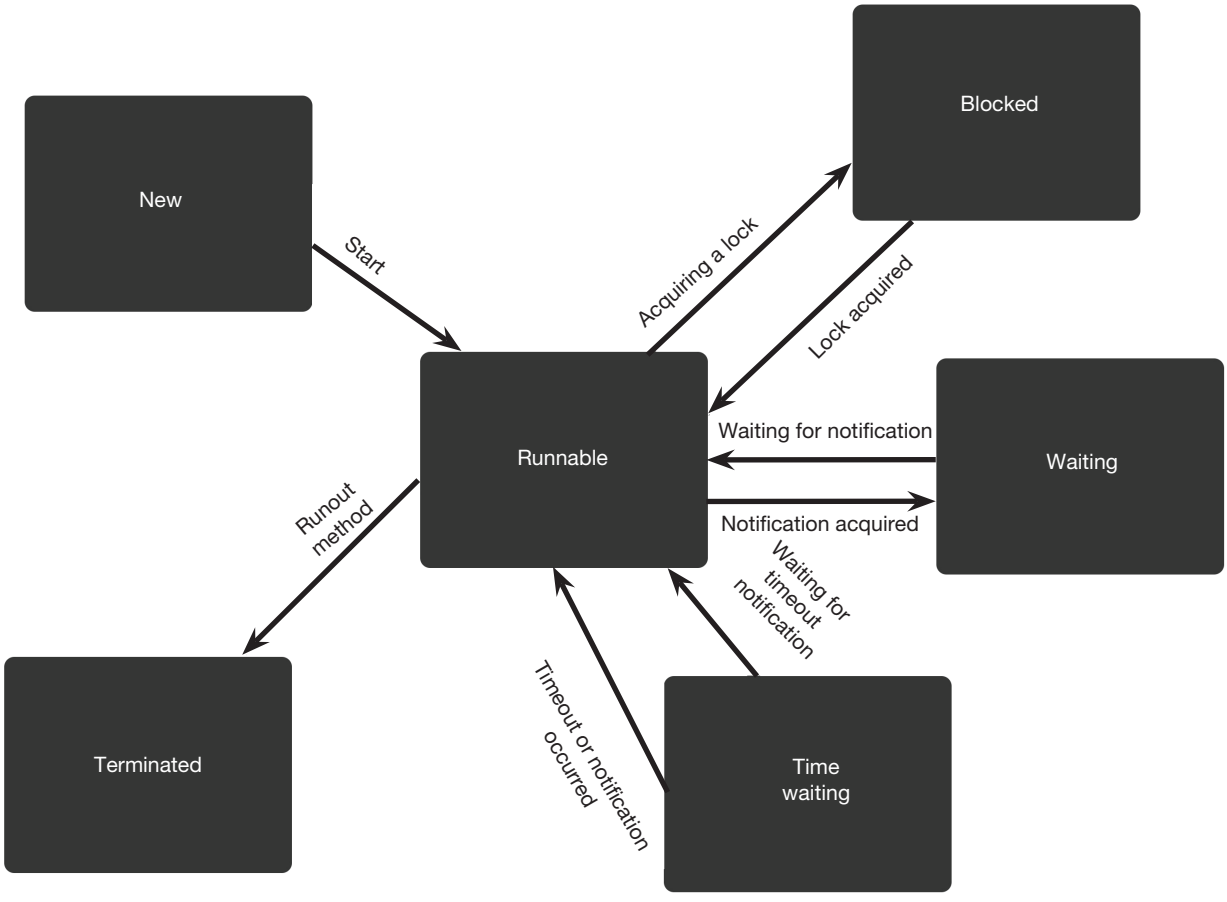

Figure 19.4 Transition of threads from different phases.

19.4.2 Programming with Multithreading

A thread is created in Java by either extending the Thread class or starting the Runnable interface. The use of the Thread class provides the required methods and constructors that contain the operations that must be performed on the objects of the class. There are several constructors such as `Thread()`, `Thread(Runnable r, String name)` that are employed for creating threads.

Methods such as `join()`, `start()`, `run()`, `sleep()`, `getPriority()`, and `setPriority(int priority)` are employed in the Thread class objects. Other methods may include testing the thread status, returning the thread id, or even

modifying the name of the thread. The `start()` method initiates a new threat and sets it up in the calling stack. Once the thread is selected for processing, it always executes its `run()` method to perform the required functionality. Here is an example of a functioning Java thread:

```
package java11.fundamentals.chapter19;
public class JavaMultiThreadingExample extends Thread {
    public void run() {
        System.out.println("My newThread is Running");
    }
    public static void main(String args[]) {
        JavaMultiThreadingExample newThread = new JavaMultiThreadingExample();
        newThread.start();
    }
}
```

The above program produces the following result.

```
My newThread is Running
```

This is a simple thread that will print the message, "My newThread is Running". It creates a single class instance with one `run()` method. This method extends the Thread class to carry out the intended functionality. Another way to perform the same functionality is to create runnable instances as presented in the following example:

```
package java11.fundamentals.chapter19;
public class JavaMultiThreadingWithRunnableExample implements Runnable {
    public void run() {
        System.out.println("My newThread is Running");
    }
    public static void main(String args[]) {
        JavaMultiThreadingWithRunnableExample myRunnableObj = new
JavaMultiThreadingWithRunnableExample();
        Thread newThread = new Thread(myRunnableObj);
        newThread.start();
    }
}
```

The above program produces the following result.

```
My newThread is Running
```

This one implements the runnable interface. Since you are not extending, you need to show the creation of an explicit Thread class object, which is th1 in this example. Another important concept is the thread scheduler. It is an important Java virtual machine (JVM) component that decides which of the runnable threads will be chosen next for execution. It can employ the method of time slicing or pre-emptive scheduling. Time slicing treats all threads equally while the pre-emptive scheduling allows for the setting of priority in the available threads.

In this regard, the `sleep()` method holds important value in Java. It pauses the thread for the mentioned milliseconds, which may be defined. The Java thread scheduler will ignore the thread and move on to the next one if it finds a selected thread with a running `sleep()` method. Here is an example of its use:

```
package java11.fundamentals.chapter19;
public class JavaMultiThreadingWithSleepExample extends Thread {
    public void run() {
        for (int i = 1; i < 4; i++) {
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
            System.out.println(i);
        }
    }
    public static void main(String args[]) {
        JavaMultiThreadingWithSleepExample myThread1 = new
JavaMultiThreadingWithSleepExample();
        JavaMultiThreadingWithSleepExample myThread2 = new
JavaMultiThreadingWithSleepExample();
        myThread1.start();
        myThread2.start();
    }
}
```

The above program produces the following result.

The sleep method throws an exception whose status can then be found with catch and then printed. There are two threads, and the running of each thread then produces a sleep period of 500ms. This means that myThread1 will get ignored in the next cycle and myThread2 will get picked up for processing. This will result in this program printing two 1s, two 2s, and two 3s as the console output.

All threads in Java can only run successfully just one time. If a thread is wrongly called multiple times, all other instances after the first one will return an illegal thread exception error. If the `run()` method is directly employed without first initiating the start, it produces the addition of the thread on the same call stack. The direct running of Thread objects is of no use, as they lose their functionality as individual threads and are simply treated as normal code objects.

Synchronization is also possible with the use of `join()` method. This is a method that stops a thread from executing until another referenced thread has been truncated. This is excellent for creating synchronized code, which may be important in several Java programs. You can also find out information about the thread which is currently running by using the `currentThread()` method. Changing the name is also possible, but it only holds value when this is used with other programming elements to produce the required functionality in a specific program.

The next important topic in this regard is to understand how the thread scheduler sets up the priority of the available threads. Java provides a priority value to every thread in a program. This is presented as a number ranging from 1 to 10. Normally, Java scheduler uses pre-emptive scheduling where the threads are arranged for execution according to these priority values. Although this may not be the case if a particular JVM is performing time slicing of multiple threads.

There are three important constants that are present in the Thread class. They describe the minimum, normal, and maximum priority for threads in the following forms:

1. `public static int MIN_PRIORITY`
2. `public static int NORM_PRIORITY`
3. `public static int MAX_PRIORITY`

The normal priority with a value of 5 is always selected for a thread as a default value. The `MIN_PRIORITY` directly sets the priority to 1 (lowest), while `MAX_PRIORITY` sets it up to 10 (highest).

An interesting point to note is that the JVM often creates a Daemon thread just to help the user created threads by providing them support and functionality benefits. Such a thread is automatically terminated by the JVM when all user threads are processed.

How will you guarantee thread priority?

Java also employs pool threads that are fixed thread objects that can be used for specific support actions. This allows for a quicker method, as creating user-defined Thread objects takes more processing time. You can also create multiple threads with a single object.

This is possible with the ThreadGroup class in Java. Multiple threads can be implemented within a single object of this class. All groups and their individual threads can be named in a customized manner. Here is an example that describes how the ThreadGroup may be set up. In an actual use, the described functionality may be complex, with each thread requiring specific processing power.

```
package java11.fundamentals.chapter19;
public class JavaMultiThreadingThreadGroupExample implements Runnable {
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }
    public static void main(String[] args) {
        JavaMultiThreadingThreadGroupExample runnable = new
JavaMultiThreadingThreadGroupExample();
        ThreadGroup myThreadGroup = new ThreadGroup("My Thread Group");
        Thread t1 = new Thread(myThreadGroup, runnable, "My First Thread");
        t1.start();
        Thread t2 = new Thread(myThreadGroup, runnable, "My Second Thread");
        t2.start();
        Thread t3 = new Thread(myThreadGroup, runnable, "My Third Thread");
        t3.start();
        System.out.println("My Thread Group Name: " + myThreadGroup.getName());
        myThreadGroup.list();
    }
}
```

The above program produces the following result.

```
My First Thread
My Third Thread
My Thread Group Name: My Thread Group
My Second Thread
java.lang.ThreadGroup[name=My Thread Group,maxpri=10]
    Thread[My Second Thread,5,My Thread Group]
```

This is a program that prints the name of the individual threads of the group as first, second, and third, and then the thread group name is returned as the Parent Thread Group after the first three threads are executed. The `list()` method then prints out the complete information of the created thread group tg1.

What will happen if two threads access the same resource at the same time?

19.5 | Concurrency

Concurrency is an important concept in computer programming. It defines the ability of a program to be executed in such a manner that any change in its running order does not affect how the final output is produced by it. It can include parallel processing of different processing units as well, which may be concurrent with each other and can be executed without affecting each other.

This improves the performance, especially in modern computers that have multiple processors with the ability to run multiple threads at the same time, which are independent of one another. The concurrency requires the decomposition of a program in partial elements so that these parts can be concurrently executed to use the available resources with improved efficiency.

Concurrency can certainly lower the program execution time and especially provide support in scenarios where an application may slow down due to a build-up of threads that may be run when performing sequential processing. All languages employ complex mathematics to create highly specialized and efficient concurrent processing schemes to improve program execution.

As the name suggests, the use of concurrency in computer programs results in various computations, which all overlap one another. The need for the sequence is eliminated, although a single program may use both sequential and concurrent program execution. Concurrency allows the application of modular programming, where the results from different computations can all be combined to produce the desired program functionality.

19.5.1 Advantages of Concurrency

The use of concurrency in Java programs can offer various benefits to programmers. Here are the top advantages that you can get with concurrent programming practices:

1. There is improvement in response of the programs. The program elements do not have to wait for specific input/output computation results, allowing the processor resources to be fully employed, while one thread is waiting for the results of a specific processing function.
2. There is improvement in the throughput of the program by a considerable margin. This is possible because parallel execution takes place constantly. This results in more tasks getting completed in every processing or execution cycle in the program. It can be simply described as a measure that shows improved resource efficiency.
3. There is improvement in the creation of better program structure. Concurrent programming is excellent for most problem domains that may require the result of several individual processes to finally reach conclusion. This may be the case where a program performs a complex function, which will often depend on the calculations of several individual elements that can be executed in any possible sequence.

Concurrent programming can use different methods. An ideal way to implement concurrent programming is multithreading. A set of threads allows an operating system or a JVM to use multiple processors to run parallel executions and therefore, gain the benefits of concurrency. Java is a language that uses explicit communication to describe the use of concurrent programming.

Java and C# are important programming languages that use communication during concurrent components that share memory resources. This is accomplished by setting up threads that can coordinate through locking mechanisms. A program that has the required functionality to avoid problems between parallel threads is termed as *thread-safe product*.

There are other methods, but we are not going to discuss them since they are not associated with the concurrency employed through specific APIs. Java implements the concept of concurrency with the use of the Thread class that we have explained above, as well as employing runnable instances which provide control over the execution of different threads.

19.5.2 Concurrency in Java

For typical computer users, they always expect that their computers are going to perform multiple tasks at the same time. They want their word processor to accept their keyboard inputs and display them on the screen, while still listening to songs that are directly streamed from the Internet. This functionality is only possible with the use of concurrent software.

Java has been supporting concurrency since its version 5.0, and the latest version is a lot more powerful. It does not provide the basic concurrency support in its JDK and class libraries, it also has an excellent collection of concurrency APIs that can perform at the highest level. The java.util.concurrent package contains powerful APIs that allows programmers to implement advanced concurrency and increase the throughput performance of their programs.

Java uses two units in concurrent programming: threads and processes. Most concurrent programming functions are delivered and performed on threads. Sometimes processes are also employed during concurrency assignments. A computer system often has a host of open processes and threads at any given time, regardless of the number of cores on the processor. Take the example of an operating system, where you can check the number of processes that are operational at the same time.

Understanding processes and threads in Java is ideal for understanding how concurrency works, especially in Java 9 update which has specific concurrency API improvements. A process is defined as a set of execution resources that have specific heap space. Processes are often defined as applications or programs, although it is possible that a single user application may be running multiple processes in the background.

The JVM mostly employs a single process, when using the computational resources of the hosting computer. However, Java provides support for creating multiple processes as well. It is possible with the use of a special ProcessBuilder object, which is a specific application beyond the scope of our concurrency article.

Threads, on the other hand, are small (lightweight) processes. They are simply termed as the most basic unit of execution that is delivered to the physical processor by JVM. Threads are also a part of the execution, but it is easier to prepare them as fewer resources are required. They always occur within a singular process in Java, while each process in any application would always have a single thread.

All threads share the resources that are allocated to the overall process in Java. This includes the heap which is assigned to the JVM as well as the open files, according to the libraries mentioned in the program bytecode during executions. Although this creates an efficient recipe, it is harder to control the use of resources without using an effective communication system for inter-thread messages.

The Java platform allows the use of multiple threads, as they are always present, even if your program only asks for one. The system also creates its own threads to perform important functions like built-in memory management. The programmer must only focus on the main thread, which is the one that contains program instructions.

Now, it is important to understand how the JVM can employ them in order to improve the performance of a Java program. Most Java applications can create multiple threads that may then be used for parallel processing or for implementing asynchronous behavior in the program.

Is there any alternative to concurrency? Explain.

Concurrency is the technique which ensures that all tasks can be speeded up by breaking them into smaller tasks that can occur independently of one another and executed in parallel in different threads. The performance of Java Runtime depends strongly on the results of the current parts that are getting processes. The Amdahl's Law governs the maximum performance gain that can be achieved with this practice as:

The *F* denotes the percentage of the program that must run in a synchronized manner and cannot go through parallel processing and *N* is the total number of processes that are running at any given time. Concurrency is not free from its own problems. This happens because threads can control their call stack, but share heap locations with each other. The first problem that concurrency produces is that of visibility while the second one is about the ideal access.

The first problem of visibility occurs when the data shared by the first thread is then used and changed by another thread without informing the first thread about it. This means that when the elements of the first thread go in their next processing, the change of data completely eliminates the functionality of the program and produces wrong data inputs within them.

The second problem of access occurs when multithreading and parallel processing is implemented during a concurrency run. This means that two different threads may attempt to access the shared data at the same time, while changing it after their particular processing in a single go. This creates a deadlock as the program cannot comply with multiple demands to access and change the same data location from two different threads at the same time.

The java.util.concurrent API package is important in this regard as it provides the support required for creating different threads and implementing strong measures for the required code synchronization. It is important that threads that are sharing data sources must run in an organized manner, where it is not possible to corrupt the data for the other thread.

This is possible with the code locks that Java implements. It ensures that several threads can run at the same time, but never use a similar call to data that can create deadlock situations. It is simply implementing with the use of *synchronized* keyword in Java. There are various benefits of using this technique in concurrency programming in the language:

1. It ensures that a singular code block is only employed by a single thread in the same timeslot. This ensures that the duplication and overwriting of data is not possible.
2. Each thread is able to view the previous modifications that occurred to the data objects in the code. This ensures that no incorrect data is processed, allowing for the removal of deadlocks and inaccurate situations.
3. It provides the blocks of mutually exclusive access to code elements. This is important as it allows threads to communicate with each other and always ensure that the shared data is current with the needs of the concurrency in Java programs.

The keyword can be easily used when defining any method in Java. This ensures that only a single thread will be able to use the code block then, during parallel execution of different program threads. The next thread that needs this code will wait until the first thread has used and left the locked method. Here is an example of its use:

```
public synchronized void myMethod()
{
    // thread critical information
    // the intended functionality
}
```

The synchronization can also be used with individual code elements that may actually be a part of a method within an object. This then creates a locked block, which is guarded by a key. The key can be created in the form of an object or a string value to create the intended lock, required for ensuring that no problems occur during concurrency. Here is an example that helps present the use of locking code sections and blocks for seamless concurrency:

```
package java11.fundamentals.chapter19;
import java.util.ArrayList;
import java.util.List;
public class SynchronizationExample {
    private List<String> wishList = new ArrayList<String>();
    private List<String> shoppingCart = new ArrayList<String>();
    public void addToWishList(String product) {
        synchronized (this) {
            if (!wishList.contains(product)) {
                wishList.add(product);
            }
        }
    }
    /**
     *
     * Now the code moves to add the next product to the shopping cart.
     * If there are no products left in the wish list, it returns Null.
     */
    public String addToShoppingCart() {
        if (wishList.size() == 0) {
            return null;
        }
        synchronized (this) {
            // Checking if any product is available in the wish list
            if (wishList.size() > 0) {
                String s = wishList.get(0);
                wishList.remove(0);
                shoppingCart.add(s);
                return s;
            }
            return null;
        }
    }
}
```

Java also provides another method to ensure that threads do not make a mistake when picking up values for the required fields. This is accomplished by using the *volatile* keyword in the declaration of the variable for any Java class. It guarantees that whenever a thread is using this attribute, it will read the most recent value for the required information. However, it does not create an exclusive lock on the variable. There are conditions where this mode of functionality is required in a program.

A volatile variable automatically updates the variables as well, if they are modified within a single thread. This means that such a variable can often work as a reference for multiple values that may get changed during a temporary case scenario. The

setter is employed in such settings to initialize and assign a value to the variable. This allows for placing an address change and allowing the stored values to be available for other threads that may attempt to use it.

Concurrency significantly depends on the accurate use of the available memory for the JDK and the JRE. The memory model controls the communication between the memory which is assigned to the individual threads and the main memory available to the entire Java program, which is allocated by the JVM when the program is running on a particular computer.

The memory model is responsible for creating and defining the rules that govern the use of memory by different threads. It also controls the way information is communicated between the different threads. It describes the solutions for keeping memory available for the program. This is produced by allowing a thread to update its use of memory from the available main memory.

Atomic operations are important in Java, as they are defined as standalone tasks that must be completed without any interference from other program tasks. It is important that atomic operations are identified during their execution. Java allows the specification of a variable, when it is running an atomic state. This is automatically possible, but operations with long or double literals must be defined as atomic by using the *volatile* keyword with their declaration.

Normal operations like increments and decrements using the ++ and – – operators are not considered atomic by Java, when performed on integer datatypes. However, by defining the value as an atomic one, it is possible to use such functionality. Java 9 onwards, Java supports the use of atomic variables that are already defined and have the necessary methods to perform different mathematical operations.

The synchronization is carefully maintained by the memory model as it updates information when it finds that a block of code is performing modification on a locked set. All previous modifications are available to the model that ensures excellent integration and the ability to simply remove all deadlock instances.

The simplest way to avoid problems with concurrency is to share only immutable data between threads. Immutable data is data which cannot be changed.

19.5.3 Concurrency Support

Java understands the importance of supporting concurrency. It provides several support methods that ensure that a class cannot be changed by proving several elements. All the fields and the class declarations are immutable when they remain finalized and their reference does not move during the construction of the class.

All fields that describe movable data objects remain private and are not used with a setter method. They are also not returned directly. Even if their value is changed within the class, it does not affect their use outside of the class in any manner whatsoever. This means that it is possible to have a fixed class that contains objects that have mutable data.

Take the example of mutable information such as Arrays that may be actually handed to a class externally, when the class is in its construction phase. The class can protect such elements by creating defensive copies of the required elements. This ensures that an object outside of the class cannot change the data in such fields.

Defensive copies play an important role in protecting your classes. This is important because other code elements can call for the class and in turn, change the data present in it in a manner not anticipated in your program's logic. A defensive copy is an excellent mechanism for ensuring data integrity. It works in a simple manner. Whenever a calling code asks for information, the immutable class creates a data copy and provides it to the code, without ever affecting the actual information stored in the immutable class. Here is an example of a defensive copy:

```
package java11.fundamentals.chapter19;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class DefensiveCopyExample {
    List<String> myArrayList = new ArrayList<String>();
    public void add(String s) {
        myArrayList.add(s);
    }
    // Following code creates a defensive copy and return the same. In this case, the
returning list remains immutable.
    public List<String> getMyArrayList() {
        return Collections.unmodifiableList(myArrayList);
    }
}
```

This program creates a new list for as a copy of the original list and passes it on to the relevant code. The original list never changes its status as array values.

Another important concept is a thread pool that holds the work queue for a Java program.

The pool keeps a record of all Runnable objects and constantly updates the list as and when required by the program. You can use the `execute(Runnable r)` code if you want the current thread to enter the queue where it will be called according to its preference that was explained earlier. The pool is named Executor and its implementation is utilized from java.util.concurrent.Executor interface. Worker threads can be easily added to the Executor, while there are also methods available to shut it down and terminate the thread processing.

Java can handle all types of asynchronous operations with the availability of this particular interface. There are different ways to implement execution tasks, such as using the Future interface. It allows asking for the results from a Callable task in Java. The CompleteableFuture option is available since Java 9 is very strong, as it ensures that all time-consuming activities are arranged in an ideal manner.

It allows the use of two approaches to provide concurrency in Java programs. The first is to ensure that all application blocks are arranged in a logical manner and follow the steps that are required to complete particular tasks. Another is to create a non-blocking approach where the application logic only moves with the flow of the tasks that are required for a program. This functionality allows the creation of different steps and stages and provides better control over the code callbacks that are required in any Java application.

19.5.4 Concurrency API Improvements

Java 9 contains considerable concurrency API improvements that occur as defined in the JEP 266 document. Here, we describe the improvements that are shared by the Java 9 documentation regarding the ability to better use CompletableFuture class and providing a lot more power in the language when compared with the older version.

The main focus of the concurrency improvements is the CompletableFuture API. The motivation behind these improvement steps is that the concepts of concurrency and parallelism must be fully integrated in the programming and execution requirement to give optimum control to a Java developer.

The improvements appear by providing better support in the relevant Java libraries, in turn adding the added functionality to the classes and the methods that may come under them, especially related to threads and concurrency settings. The interface improvements are also based on creating Reactive Streams that use the principles of concurrency and parallel thread processing.

Reactive Streams can provide better support through the use of the class *Flow*. It allows publishers to create items for different subscribers. The individual solutions can all use simple communication with the use of an information flow control to provide the ability to react to a dynamic program execution environment. A utility class is also added in the form of SubmissionPublisher that allows developers to create customized components capable of independent communication.

The main enhancement remains on the CompletableFuture API that was already discussed. Time-specific enhancements were certainly required during concurrency operations in Java as they had the ability to ensure consistency and make sure that all deadlock situations can be eliminated with accuracy.

There are excellent methods that are added to control the duration of different threads and their relevant operations. An Executor is added, which is extremely powerful and provides the use of a static method, such as delayedExecutor. It is a powerful method with the capability of providing a time gap between the reading of the code and the execution of a particular task.

When combined with the Future functionality, it can create excellent support for complete threads that can use concurrency, but still provide the reactiveness necessary for enjoying the ideal support for improved parallel processing. There are several small improvements as well, which may not be apparent to a normal programmer, but significantly improve the capacity of handing multithreading that ensures reactive programming. This is built within the concurrency structure to offer improved throughput while maintaining greater control over enhanced heap sizes and shared memory spaces.

The overall concurrency improvements create a significant difference between the reactive programming capability of Java 8 and Java 9. The new programming kit employs the Flow API at its maximum power by providing four static interfaces. These interfaces contain all the methods that can provide a flow in program executions, where the publisher can control the production of data objects according to their consumption by different consumers of the program.

The production is handled by a Publisher, which produces the data required and received by different subscribers. The Subscriber acts as a receiver, where its argument depends on the definition presented by Subscription, that links it with the Publisher. Both parties combine together to create the Processor, which specifies a specific transformation requirement.

The data streams can be set up with the use of the Publisher by using the `subscribe()` method. It is used to connect a Subscriber with a Publisher. If a Subscriber is registered already, then this method registers an error and returns an illegal exception.

A Subscriber works to provide callback code for data items through the Flow library. It can be declared with `onSubscribe()` method. However, the same declaration can also occur with `onError(Throws exception)`, `onComplete()`, and `onNext(Item)`. The first method describes a registration from allowing the requests for new data items to be moved to the subscriber. This situation is required for implementing the program flow measures that provide concurrent behavior.

There are also new methods that Java 9 update brings in the CompletableFuture. They allow the creation of different stages when performing concurrency functions. This ensures that the program always remains protected from sudden failures. A method file `completedStage()` and `failedStage()` are perfect for providing information that are in the processing phase. They can throw exceptions and provide information that may be useful for showing that a particular function is completed, ensuring the use of aggressive concurrency practices in the program.

The CompletableFuture is now more powerful, especially due to the support of delay and timeouts which are ideal for use in a large and complex program that often employs multiple threads and thread groups. The delay is an excellent choice for use in concurrent applications, where it simply associates the required time that significantly covers a thread from corrupting the available data values.

There is another method called `Timeout()`. It is excellent for ensuring that a particular future situation is eliminated before a relevant code must run. This situation may not always be required and therefore, the method only throws an exception if it is required for a `CompletableFuture<>` functionality. However, this functionality can be further improved by experimenting around and creating different logical approaches to achieve the desired functions.

Here is an important example of how the delay can be presented using different Flow controls:

```
package java11.fundamentals.chapter19;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
public class DelayedExecutorExample {
   public static void main(String[] args) throws InterruptedException,
ExecutionException {

       CompletableFuture<String> completableFuture = new CompletableFuture<>();

       completableFuture.completeAsync(() -> {
          try {
             System.out.println("CompletableFuture - Executing the code block");
             return "CompletableFuture completeAsync executed successfully";
          } catch (Throwable e) {
             return "In catch block";
          }
       }, CompletableFuture.delayedExecutor(3, TimeUnit.SECONDS))
             .thenAccept(result -> System.out.println("In Accept: " + result));

       for (int i = 1; i <= 10; i++) {
          try {
             Thread.sleep(1000);
          } catch (InterruptedException e) {
             e.printStackTrace();
          }
          System.out.println("Executing For Loop Block : " + i + " s");
       }
   }
}
```

The above program produces the following output.

```
CompletableFuture – Executing the code block
In Accept: CompletableFuture completeAsync executed successfully
Executing For Loop Block :  1 s
Executing For Loop Block :  2 s
Executing For Loop Block :  3 s
Executing For Loop Block :  4 s
Executing For Loop Block :  5 s
Executing For Loop Block :  6 s
Executing For Loop Block :  7 s
Executing For Loop Block :  8 s
Executing For Loop Block :  9 s
Executing For Loop Block :  10 s
```

This is a simple program that will present the running outside three times with values of 1, 2, and 3 seconds. The inside future will represent the screen output of processing data while presenting the required acceptance. The delay will then be overproducing the remaining 4, 5, 6, 7, 8, 9, and 10 seconds' values.

Methods that were not used in the previous version are removed and new methods are added in Java 9, which provide excellent support to the ever-improving concurrency API present in the programming environment. It includes improvement in the atomic functions that often employ reference array methods and Boolean comparison functions.

One problem that may appear with the Flow API is that sometimes, Publishers can produce data at a much faster rate, compared to its consumption by multiple Subscribers. These situations require the creation of an ample buffer, which can hold unprocessed elements. These elements produce backpressure, which Java 9 handles with the creation of logic that removes the collection of elements with various reactive programming techniques.

Reactive programming, when introduced with multithreading, truly empowers Java 9 and allows developers to use it at the best of their capacity. The development environment is still evolving to produce more API improvements. With the main improvement available in the Java package, it is inevitable that different developers may end producing better API support models that will help in using reactive programming principles with improved concurrency.

A conclusion to the concurrency API improvements in focuses on describing how the use of effective evolution is required for using stronger techniques and methods such as multithreading. With improvement over the control of functions that are related to one another, it is possible to use the full advantages of concurrency as governed by its theoretical limit.

With the tools of CompletableFuture and new improvements like ForkJoinPool and other relevant classes, it is important to understand the individual threads in Java and how they can be set up to ensure the best use of multithreading and the ideal parallel processing with the help of parallel programming principles.

There are other API improvements as well. One important class in this regard is the ConcurrentHashMap, which is designed to support a system of concurrency retrievals. It is an excellent design capable of providing the Hash table functionality in an improved manner. It has the same methods, but all with improved performance. It is a class that does not employ locking, but offers all functions to remain safe from thread deadlock issues and other parallel processing problems.

The mapping class works well with retrieval and ideally allows the use of updating within the same processing zone due to not locking the code or data. Different parameters of the hash table can also be retrieved, allowing the use of enumeration or the iterator. The hash table has improved control and can also be resized if it faces a certain load. Size estimates are possible as well, by using the initialCapacity constructor. It is certainly a class that allows for improved concurrency in Java 9 when compared with the older JDK environments.

The `newKeySet()` method is available for setting the mapped values or simply recording the different positions. There are few differences from other classes, as it does not allow the use of null value. Concurrent hash maps are excellent for use when combining serial and parallel operations. They offer safe way to apply concurrency, while still ensuring that the ideal arguments can be used in the Java program.

ConcurrentHashMapgenerates a frequency map as well where it can produce a histogram of usable values. This means that it can support functions both in an arranged manner, while still containing a set of parallel executions that may be controlled with a single forEach action. Remember, the elements should always be used in a manner as to not get affected if the order of the supplied functions is changed since it will remain random, during bulk operations.

There are other operations of mapped reductions and plain reductions in this class. Plain reductions do not correspond to a return type, while mapped reductions are used for accumulating the application of functions. Sequential methods occur if the

map returns a lower size than the given threshold. The use of `Long.MAX_VALUE` provides suppression control on parallelism, while the use of 1 acts as allowing for maximum parallel results. This is possible because the program creates subtasks by using the `ForkJoinPool.commonPool()` method that allows parallel computations. The ideal programming starts by picking one of these extremes and then revising the values to achieve your required level of overheads against the delivered throughput.

The hash map can speed up the executions using parallel processing, but it is not always the preferred solution when compared to sequential processing. If small functions that are placed on separate maps are used, they will often execute slower than serial transformations. Parallel processing will also not be valuable if it is simply taking care of different tasks that are not related to each other, and do not produce a net gain over the normal capacity of the Java compiler.

The ConcurrentHashMap can also be created in the image of another map. This is an excellent option when you are experimenting with their use and have not yet identified the ideal approach to use the available HashMap options.

There are some excellent parallelism options since Java 9, such as the ability to use the `newWorkStealingPool()` method. This is a method present in the Executor class and allows the creation of a threading pool. It uses the available processes as the value for parallelism, and this defines the use of a process where all applicable processors are working simultaneously on different tasks. If the parallelism value is inserted in this method, then the thread pool keeps the required number of threads for the parallelism. The class then uses the creation of different queues to ensure that multiple threads are not in contention for the limited execution slots.

19.5.5 Dealing with Individual Threads

Let us once again discuss the individual threads produced by the JVM. Normally, the `main()` method creates a single thread on JVM. The thread continues to perform until the exit method of runtime class is called. The program ends when all non-daemon threads have performed their functions and have already been recalled. Another way for individual thread elimination is through an exception.

Individual threads can have several important parameters. It includes the name, the target, and the stack size available to the thread class object. The creation of individual threads is required especially when you want to implement the strong power associated with multithreading capacity of Java compiler.

There is excellent control available over the individual threads, with the `currentThread()` method allowing you to reference the currently processed object. There are other methods available as well, such as `yield()` which ensures that the thread is willing to drop its current use of processing for any other thread that is present in the processing scheduler.

The individual threads can also be controlled by momentarily making the caller unavailable for processing. Invoking a method like `onSpinWait()` is ideal for situations where you may want a loop construct that needs to show that the calling thread is, in fact, busy waiting for information from other parts of the program. This method keeps spinning unless controlled by a particular process or exception produced using the available flags. Here is an example:

```
package java11.fundamentals.chapter19;
public class OnSpinWaitExample {

    volatile boolean notificationAlert = true;

    public static void main(String args[]) {
        OnSpinWaitExample onSpinObj = new OnSpinWaitExample();
        onSpinObj.waitForEventAndHandleIt();
    }

public void waitForEventAndHandleIt() {
while ( notificationAlert ) {
java.lang.Thread.onSpinWait();
System.out.println("In While Loop");
}
processEvent();
}
    public void processEvent() {
        System.out.println("In Process Event");
    }
}
```

The above example produces the following result.

```
In While Loop
In While Loop
In While Loop
In While Loop
In While Loop
In While Loop
In While Loop
In While Loop
In While Loop
In While Loop
In While Loop
In While Loop
```

The control over the individual threads is still available if they are grouped together. You can get the count of the threads, as well as learn about currently active threads. However, remember that it is possible to perform multithreading by allowing them to process in a concurrent manner, where they do not cause an interruption because of the shared memory space.

Another important concept to understand about individual threads is their ability to perform communication, which is termed as co-operation in Java. It is a method of allowing one thread to pause while ensuring that another thread can be executed in a particular order. It is accomplished using `wait()`, `notify()`, and `notifyAll()` methods that belong to the Object class in Java.

Let us start with `wait()` as it is a method that causes the currently operational thread to release its data lock and go into the waiting mode. The waiting can be for a defined period or only returned with the use of the notify methods. A time period can be mentioned for the return as the method argument or left without use for the following methods.

1. **`notify()`:** This method causes a single thread to come out of the waiting stage and become active on the current object. If there are multiple threads that are present within a single object, the selection of the awakening thread is random and lies at the discretion of the system components.
2. **`notifyall()`:** This method is excellent when you want all threads that are present in a particular object to come out of their waiting stage. This is important since it removes the random nature of the previous method and allows for better concurrency functions.

Remember, the `wait()` method is different from `sleep()`. The former method applies on the Object class while the latter works on the Thread class objects. The `wait()` method remains a non-static method and can be revoked with the use of `notify()` or `notifyall()` methods, as well as specific time. On the other hand, `sleep()` works with a specific amount of time, and does not provide the dynamic control which is required when actively performing aggressive reactive programming that takes full advantage of the capabilities of the Java compiler.

19.5.6 Synchronizing Code Blocks

Synchronizing code blocks is an excellent way of creating particular methods and then controlling them to work in an organized manner. Synchronization can also occur in a limited capacity, where we can place some part of a method within a synchronized block while leaving the rest of it for random execution.

It is in fact, a way of locking parts of code that you do not want to be accessible to any other resource. Remember, the scope of using a synchronized block is always smaller than a complete method and should always be employed in this manner. Here is an example of a synchronized block used within a Java program:

```
package java11.fundamentals.chapter19;
public class SynchronizationBlockExample {
    public static void main(String args[]) {
        Calculator calculator = new Calculator();
        WorkerThread1 t1 = new WorkerThread1(calculator);
        WorkerThread2 t2 = new WorkerThread2(calculator);
        t1.start();
        t2.start();
    }
}
class Calculator {
    void multiplicationTable(int n) {
        // Following block will ensure the method is accessible in synchronized manner
        synchronized (this) {
            for (int i = 1; i <= 10; i++) {
                System.out.println(Thread.currentThread().getName() + " : " + n * i);
                try {
                    Thread.sleep(400);
                } catch (Exception e) {
                    System.out.println(e);
                }
            }
        }
    }
}
class WorkerThread1 extends Thread {
    Calculator t;
    WorkerThread1(Calculator t) {
        this.t = t;
        this.setName("Worker Thread 1");
    }
    public void run() {
        t.multiplicationTable(3);
    }
}
class WorkerThread2 extends Thread {
    Calculator t;
    WorkerThread2(Calculator t) {
        this.t = t;
        this.setName("Worker Thread 2");
    }
    public void run() {
        t.multiplicationTable(40);
    }
}
```

The above program produces the following result.

```
Worker Thread 1 : 3
Worker Thread 1 : 6
Worker Thread 1 : 9
Worker Thread 1 : 12
Worker Thread 1 : 15
Worker Thread 1 : 18
Worker Thread 1 : 21
Worker Thread 1 : 24
Worker Thread 1 : 27
Worker Thread 1 : 30
Worker Thread 2 : 40
Worker Thread 2 : 80
Worker Thread 2 : 120
Worker Thread 2 : 160
Worker Thread 2 : 200
Worker Thread 2 : 240
Worker Thread 2 : 280
Worker Thread 2 : 320
Worker Thread 2 : 360
Worker Thread 2 : 400
```

This program uses a synchronized section within a single method of `multiplicationTable()` which includes a counter that then produces a sleep delay for the thread. This situation results in generating a set of numbers where multiples of 3 are printed 10 times, while the same is then repeated with multiples of 40. Similar functionality can be obtained by using an anonymous class where you do not have to define it separately for the program operations.

There are static methods as well, which only use fixed information. Using synchronization for such methods ends up locking the entire class, rather than a particular object that calls for the method. Take the example of two objects from the same class that share information.

They may be termed as ob1 and ob2. The use of synchronized methods will ensure that interference is not possible between different actions. The use of static synchronization is excellent because it ensures that the lock is available for the class and away from the individual objects. The lock is easily created on the class by using this method:

```
static void multiplicationTable(int n) {
       // Following block will ensure the method is accessible in synchronized manner
       synchronized (this) {
          for (int i = 1; i <= 10; i++) {
             System.out.println(Thread.currentThread().getName() + " : " + n * i);
             try {
                Thread.sleep(400);
             } catch (Exception e) {
                System.out.println(e);
             }
          }
       }
}
```

This is the declaration which can be available on the class when used with the static keyword at the top with a defined class method.

19.6 | Understanding Deadlock

Java 9 update brings excellent multithreading support, which certainly brings deadlock into the discussion. Remember, a deadlock is an unavoidable reality when performing multithreading. It is the inevitable result of a situation where threads must wait for locked objects and work according to the defined thread rules. Figure 19.5 shows a visual representation of deadlock.

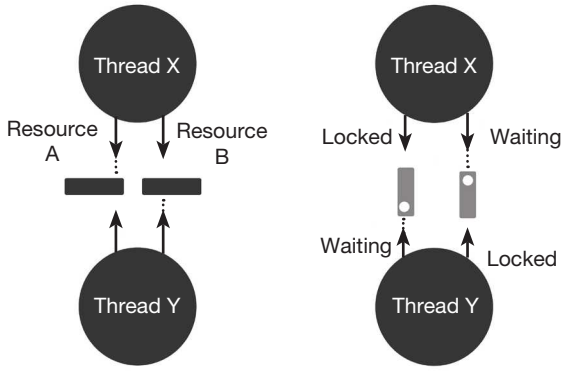

Figure 19.5 Visual representation of deadlock.

The deadlock is a natural situation that occurs in multithreading, when multiple threads are waiting for each other to release the lock on the object that needs to be processed, as the following example elaborates:

```
package java11.fundamentals.chapter19;

public class DeadlockExample {

    public static void main(String[] args) {
        final String firstResource = "First Resource";
        final String secondResource = "Second Resource";

        // Following code demonstrates thread 1 attempt to lock firstResource then
secondResource
        Thread thread1 = new Thread("First Thread") {
            public void run() {
                synchronized (firstResource) {
                    System.out.println(this.getName() + " : First Resource is Locked");
                    try {
                        Thread.sleep(100);
                    } catch (Exception e) {
                    }
                    synchronized (secondResource) {
                        System.out.println("Second Resource is Locked");
                    }
                }
            }
        };
        // Following code demonstrates thread 2 attempt to lock secondResource then
firstResource
        Thread thread2 = new Thread("Second Thread") {
            public void run() {
                synchronized (secondResource) {
                    System.out.println(this.getName() + " : Second Resource is Locked");
                    try {
                        Thread.sleep(100);
                    } catch (Exception e) {
                    }
                    synchronized (firstResource) {
                        System.out.println("First Resource is Locked");
                    }
                }
            }
        };
        thread1.start();
        thread2.start();
    }
}
```

The above code produces the following result.

```
Second Thread : Second Resource is Locked
First Thread : First Resource is Locked
```

This is an excellent demonstration of the deadlock situation. The first synchronization in thread 1 already sets first resource to locked, which is then suspended by using sleep. This means that the program proceeds to run thread 2, which produces the screen output about second resource, while once again sleeping to allow thread 1 to run.

However, the program is now stuck due to deadlock because thread 1 must wait for second resource to be free of lock in thread 2, which means that the next thread for processing is thread 2. The thread 2 faces a similar situation as first resource is similarly locked by the thread 1, creating a condition where the program is logically stuck.

19.6.1 Resolving Deadlock

Many programmers choose to simply ignore that it is possible to have a deadlock condition during the execution of a Java program. They believe that a deadlock may only appear due to a poor programming effort, and should always be removed during the planning phase of the program. They also believe since a deadlock can freeze a program, a simple solution would be to restart the application, resulting in a new state where the same deadlock condition may never appear again.

Obviously, this is the wrong way to go about Java programming as you must ensure from your end that a program is free from any conditions that may produce a stall or freezing, especially if it is providing control and support for a large-scale operation. Another way to go about it is to build a detection system in your program.

This is achieved by adding a special task that gets executed only to check the current status of the program parameters. This ensures that it is possible to detect, if the program is entering a deadlock situation, due to the reasons that we just described above. This is possible by checking whether tasks are stuck in their current functionality. The remedy can occur by eliminating a stuck task or forcefully liberating a shared resource, which is required for other program elements to function normally.

Another method is to produce a prevention of the Coffman conditions. These are the four ways in which a deadlock can occur during the program execution. A program can prevent a condition where special measures are built within the program structure that stop the occurrence of these four situations.

Another method to avoid deadlock situation is to make sure that your program design avoids deadlocks. This is possible by ensuring that your program first obtains the information about the required shared sources each time a particular task starts in your program. This ensures that there is always a set of available resources that allow your task to get executed without any problem. If there is a lack of resources, you can insert conditional delays that will always ensure that only those starts are initiated that can be completed in the current set of available resources.

19.7 | Concurrent Data Structures

Java 9 update brings excellent data structure options, but they are never designed to provide the ideal support for concurrent operations. The use of an external method ensures that ideal synchronization is possible. However, it significantly increases the computing time of your application. There are certain data structure practices that you can employ that will allow you to create data structures that are fully supported by Concurrency API. There are two groups of these structures – blocking and non-blocking data structures.

The blocking structures are present as methods that ensure that the calling task is blocked when you are employing the data structure although it does not have the intended values. On the other hand, non-blocking structures do not block the calling tasks. These methods are designed to throw an exception if a task calls on the data structure when it does not hold value. It can also produce a null value as a result.

19.8 | Multithreading Examples

Multithreading Java examples can be best followed by understanding the thread pool in Java. It is a collection of Runnable objects. Worker threads in it are controlled by the Executor framework and create a life cycle where it maintains the overall life of the Java program. It is possible to use:

```
Executors.newSingleThreadExecutor()
```

This is a method that will create a single thread, but one which contains multiple running items. Here is how it can be applied:

```
package java11.fundamentals.chapter19;
public class RunnableExample implements Runnable {
    private final long counter;
    RunnableExample(long counter) {
        this.counter = counter;
    }
    @Override
    public void run() {
        long total = 0;
        for (long i = 1; i < counter; i++) {
            total += i;
        }
        System.out.println(total);
    }
}
```

Once the thread is established, it can be executed in numerous ways to gain the required facilities. We have significantly discussed the concept of Future. Now we describe that the Callable object can also be employed with concurrent processing. When this is used with an executor, it returns an object of the Future type. The `get()` method can be employed to receive a Future object which can then be employed for designing the ideal concurrent processing schemes.

An excellent use of thread control is possible when it is mixed with the ideal use of the `sleep()` method. Defining subclass is another effective method when implementing new threads. These are instances where you can control individual thread instances to gain better performance.

Another key way in which multithreading is performed is by using the Swing application. This is created through a set of conditions where there is an initial thread that includes the `main()` method, which is designed to exit at the occurrence of a defined event. This situation then gives rise to another event dispatching thread (EDT), which runs the specific functionality required from the program.

This technique provides excellent control over a program which may perform better with concurrent behavior. Multithreading is excellent when it can be divided in the form of different tasks that can then be set up according to the occurrence of key events during program execution. The main program then becomes a controller which keeps the operations shifting to the required threads. Remember, there is always the need to have a background (daemon) thread available, which takes care of the intensive tasks and the input/output controls that you need in a program environment.

When threads are switched according to their functionality, it is possible that application users may identify a pause between their inputs and the response produced by the application. This means that the EDT, which is set up in the program must never work for over 100 milliseconds, where it may start to produce a noticeable delay in the functioning of the worker threads.

Multithreading problems can be avoided in Java programs when the problems that are required for controlling the graphical user interface (GUI) are also performed on the EDT. This ensures that all operations remain thread safe, and can be carried out without ever causing any problems.

On the other hand, tasks that include the input/output processes should be kept on the main thread as they need to provide swift interaction, otherwise the program behavior is slow and does not produce the intended benefits. Another capacity for use is that Swing operations can be updated with the use of a delay timer. This allows the thread to update its components at controlled time intervals, resulting in the generation of the required information.

Another capability is with the publishing of thread information. This can happen with the use of:

```
protected final void publish(V... chunks)
```

The above method, which should be called inside the `doInBackground()` method, sends data to the process method to deliver intermediate results

```
protected void process(List<V> chunks)
```

The above method works in an asynchronous manner to receive datasets from the `publish()` method.

This is a program that produces the intermediate results. The method can be useful for implementing improved controls in the program. It allows Java programs to produce the situation of the current tasks. This allows for the implementation of practices that result in a better control over the data streams that are available in the program.

19.8.1 Matrix Multiplication

A common problem which you can use to learn concurrent programming is to create a matrix multiplication program. You can learn the simple concept of matrices where it is possible to multiply two matrices, as long as the number of columns of the first matrix is equal to the number of rows of the second matrix. This is termed as having matrices $A(p \times q)$ and $B(q \times r)$ where the first value in the parentheses depicts rows and the second value describes columns.

This operation is possible using different schemes. The Java library recognizes this need as an important function and provides the *MatrixGeneratorExample* class for these operations. This class uses the following code for its operation:

```
package java11.fundamentals.chapter19;
import java.util.Arrays;
import java.util.Random;
public class MatrixGeneratorExample {

    public static void main(String args[]) {
        System.out.println(Arrays.deepToString(generateMatrix(3,3)));
    }

    public static int[][] generateMatrix(int rows, int columns) {
        int[][] matrix = new int[rows][columns];
        Random random = new Random();
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                matrix[i][j] = random.nextInt() * 10;
            }
        }
        return matrix;
    }

}
```

The above program produces the following result.

```
[[853339708, 1266531654, -21594734], [1024418966, -490434492, -2020854294], [1604225446, -2030454084, 568464094]]
```

As we can understand from the structure of this class, it is easy to perform this calculation using a serial method. This will be a method where we will use two matrices and then use loops to multiply the elements that are present in the rows and columns

of the matrices. The results will be updated each time when calculating the value, and the loop will end when all elements are multiplied.

You should always produce a sequential version of every algorithm before you employ parallel versions. Here is the sequential method for multiplying matrices:

```
package java11.fundamentals.chapter19;
import java.util.Arrays;
import java.util.Random;
public class MatrixSerialMultiplierExample {

    public static void main(String args[]) {
        int[][] firstMatrix = generateMatrix(3,3);
        int[][] secondMatrix = generateMatrix(3,3);

        System.out.println(Arrays.deepToString(multiplyMatrix(firstMatrix, secondMa-
trix)));
    }

    public static int[][] generateMatrix(int rows, int columns) {
        int[][] matrix = new int[rows][columns];
        Random random = new Random();
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                matrix[i][j] = random.nextInt() * 10;
            }
        }
        return matrix;
    }

    public static int[][] multiplyMatrix(int[][] matrix1, int[][] matrix2) {

        int row1 = matrix1.length;
        int column1 = matrix1[0].length;
        int column2 = matrix2[0].length;
        int[][] result = new int[row1][column1];

        for (int i = 0; i < row1; i++) {
            for (int j = 0; j < column2; j++) {
                result[i][j] = 0;
                for (int k = 0; k < column1; k++) {
                    result[i][j] += matrix1[i][k] * matrix2[k][j];
                }
            }
        }
        return result;
    }

}
```

The above program produces the following result.

```
[[-59142824, -631908344, -1314105512], [-341256416, 293358376, -1805709024], [194502204, -108105864, 1115015312]]
```

This is a serial program that uses three matrices. The first two matrices are multiplied using the basic matrix multiplication rules and the results are stored in the `result[][]` matrix. This program implies that both matrices are fit for performing the mathematical operation. This can be checked using other programming elements. We can also check the execution time by using a random generator to multiply large enough matrices, like ones with 2000 rows and columns, to get how many milliseconds it takes to reach the result. Here is a small section which shows how you can record time for the matrix multiplication:

```
package java11.fundamentals.chapter19;
import java.util.Arrays;
import java.util.Date;
import java.util.Random;
public class MatrixSerialMultiplierWithTimeExample {

    public static void main(String args[]) {
        int[][] firstMatrix = generateMatrix(3,3);
        int[][] secondMatrix = generateMatrix(3,3);

        Date start=new Date();

        System.out.println(Arrays.deepToString(multiplyMatrix(firstMatrix, secondMatrix)));

        Date end=new Date();

        System.out.printf("Total Time Taken : %d milli seconds", end.getTime() - start.
getTime() );
    }

    public static int[][] generateMatrix(int rows, int columns) {
        int[][] matrix = new int[rows][columns];
        Random random = new Random();
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                matrix[i][j] = random.nextInt() * 10;
            }
        }
        return matrix;
    }

    public static int[][] multiplyMatrix(int[][] matrix1, int[][] matrix2) {

        int row1 = matrix1.length;
        int column1 = matrix1[0].length;
        int column2 = matrix2[0].length;
        int[][] result = new int[row1][column1];

        for (int i = 0; i < row1; i++) {
            for (int j = 0; j < column2; j++) {
                result[i][j] = 0;
                for (int k = 0; k < column1; k++) {
                    result[i][j] += matrix1[i][k] * matrix2[k][j];
                }
            }
        }
        return result;
    }

}
```

The above program produces the following result.

```
[[676643992, 1091729664, 1847180544], [1048256660, -2143242596, -332108212], [-1842184312, -1674877996, 974596884]]
Total Time Taken : 1 milli seconds
```

This section of the code will first record the time before the start of the matrix multiplication and then also record the time when the final calculation has occurred. The output to the console is done by subtracting both times; this results in the number of milliseconds it took to complete the operation.

Now that we have learned how to carry out this program, it is time to prepare concurrent versions to see the difference. Sequential programming often has only one avenue, but there are various methods for parallel processing. You can use a single thread for each element when creating the result matrix. You can also employ an individual thread for every row for the same matrix. Another effective way of concurrent programming is to employ all the threads that are available in the current JVM structure.

The first method creates too many threads, as the number of elements are too great for large matrices. Take the example of 2000 × 2000 matrices. Multiplying these matrices will result in the form of a matrix that will require the calculation of 4,000,000 elements. Although this can be accomplished, we will not be executing this program in this book. However, we will represent the other two options here.

19.9 | Designing Concurrent Java Programs

There are several ways to create a concurrent Java program, one that employs the best principles of parallelism, while using the different resources and tools that are present, especially in the latest Java development environment. Here is a strategy that you can use to ensure that you always end up with the required concurrent algorithms for use, regardless of the actual functionality that you need in the program.

A good practice in this regard is to always create a sequential version of the algorithm that you want. This is important as it allows you to measure the advantages of using concurrent code, as well as understand whether both versions are producing the same program results, which is important in all applications.

The creation of an initial sequential program will allow Java programmers to later compare the throughput of both program versions. This will ensure that you have hard numbers that describe the results of the responsiveness of the program, as well as judge the amount of data that both algorithms processed, as the overall output is always the same.

1. The first step in creating the ideal concurrent algorithms is to analyze the parameters and the delivered performance of the sequential version of your program. Special attention must be given to sections that especially took a lot of time or used a significant portion of the available heap to the Java program. Some ideal options in this regard are the loops and decision-making elements in the program that can take a lot of time before producing the ideal results.

   Other analytical elements include looking at independent parts of the program, which would not get affected, even when using concurrent programming. It may also include the object initializations and setting up the initial variables and data elements that are required for a Java program.
2. The second step in this process is to perform the concurrent designing phase. This is possible when you have analyzed your sequential code and can create a logical picture of what parts are independent and can be truly initiated with parallel program processing. There are two ways of designing concurrency. The first is to decompose the program in the form of independent tasks that can occur simultaneously. The second is the decompose the program in the form of independent data elements. You can create datasets that share resources, where you create protected access to ensure that all important data remains locked before a critical section is executed.

   Always remember that the aim of your concurrent program is to create a situation where it is important to produce a benefit. If you find that adding more functional elements will create a situation where parallel execution will not be feasible, you can choose to split the program elements to make sure that the benefits are significantly available.
3. The third step to performing concurrent algorithms in Java is to carry out the implementation using the available thread, parallelism, and concurrency options. Java, with all the tools that we have mentioned in this article, is perfectly suited to provide this option using different thread classes and the ideal Java library tools.
4. The fourth step in this practice is to test your concurrency supporting code. The parallel algorithm must be tested against your sequential code, where you must compare the important results from both avenues. We will later describe how you can monitor concurrent Java applications for performing the ideal tests that deliver useful information. The testing phase will often include debugging as well, where you simply remove any programming errors.
5. This moves us to the fifth step of tuning your concurrent program. This step may not be always required, as you should only perform customized tuning if you do not get the intended benefits from your parallel processing practices. There are some important ratios that you can use when measuring the performance of concurrent applications, such as the speed up metric. This is a ratio between the execution time of the sequential program to the execution time of the concurrent program. It should always return a value greater than one, showing that there is a time benefit for using concurrent processing. The Gustafson's Law is employed when creating parallel designs that have the ability to use multiple cores for different input datasets. This is applied with the formula:

$$\text{Speedup} = N - P * (N - 1)$$

where *P* represents the percentage of the program code which can be parallel-processed without affecting the program integrity and *N* denotes the number of available cores which can all function with their separate datasets at the same time.

There are some other points that you should also keep in mind when using parallel processing with the ability to react to the real-time application needs. Remember, not all algorithms are empowered with the use of parallel processing. Programs where it is possible to run several independent threads at the same time are ideal for implementing concurrency and multiple threads for achieving the best operational performance.

There are some important points that you should always consider before using concurrent Java tools in your programs. The efficiency of your parallel program must be significantly greater than sequential processing. This is shown by either a smaller running time or the ability of the concurrent program to process more data in the same timeframe.

Your algorithm using reactive programming elements must still focus on simplicity. Remember, the final Java program should always have the simplest form, which allows for easy testing, maintenance and other programming steps, which are required before the application is ready for live use. Portability is also important where your parallel programming solution must provide the same benefits over a number of varying platforms.

The last important element is that your parallel program must have the factor of scalability. Your program should provide the same or even greater benefits when the number of available sources for the program are increased by a considerable margin. It should always be possible to scale up and create a global application from your specific code.

19.9.1 Ideal Tips for Concurrent Java Programs

Good concurrent programs are possible when you, as a Java developer, understand your core objective. This objective is to employ concurrency as a way to enhance the performance of your application, while ensuring that it offers significant throughput benefits. Remember, you can always learn about the code and the syntax of Java. Your learning focus should always be on understanding the benefits of employing concurrency in various Java application requirements.

Some standard practices can help ensure that you produce the optimal performance from the use of concurrency. We will describe some important concurrency tips and tricks in greater detail in the following subsections.

19.9.1.1 Never Assuming Thread Order

If you do not set up synchronization schemes, you can never be sure of the order of the execution of the different program threads in concurrent programming. The thread scheduler of the specific operating system decides which task is executed first if there are multiple available options, with no order specified with the use of localization.

Assuming this will never cause an issue leaves your program with conditions that give rise to data race conditions as well as thread deadlocks. There are several applications where the exact order of the tasks in the algorithm can affect the final result of the program. You cannot be sure of the reliability of your code in such a situation. With the right synchronization methods and schemes in place, you can make sure that your concurrent program always behaves in an intended manner.

19.9.1.2 Building Scalability Option

The main objective of your concurrent program is to take the maximum advantage of the available computer resources. This includes the available memory and the processing cores. However, computing elements may not remain the same as the supporting hardware is subject to an upgrade, especially if your client decides to scale its business project.

This means that you should build scalability in all your programs. Scalable programs are great as they provide a time proof design for your clients. Good Java programmers ensure that they never assume the number of available cores or heap sizes. The ideal program would always have a reactive programming element in it, which always finds out the available resources and then implement them in a maximum capacity in the different program functions.

This is possible with the method of `Runtime.getRuntime().availableProcesssors()`, which returns the number of available processors. You can then set up your program to make use of this information in a reactive manner. This practice may increase your program overhead slightly, but it offers amazing scalability advantages, where your program can enjoy improved use of the available resources.

Designing scalability can be difficult if you perform concurrency with the use of task decomposition, rather than setting data decomposition schemes. The independent tasks can then mean that the overhead will be great because of the required

synchronization methods. The overall application performance will actually go down if your program needs to process the need for using the available processors. Always study whether you can create a dynamic algorithm which makes use of dynamic situations. Otherwise, only build limited scalability where the overhead should never exceed the advantages that can be gained from the use of additional processors.

19.9.1.3 Identifying Independent Code

The best performance from concurrency is only possible when you have identified all the independent tasks in your program. You should not run concurrency for tasks that depend heavily on each other, as this will simply require too much synchronization, negating the benefits of parallel processing. These tasks will run sequentially and the additional code will simply place a burden on the program. However, there are other instances where a particular task will depend on different prerequisite functions that are all independent of each other.

This situation is great for concurrency as you can run all the prerequisite execution in parallel and then place a condition for synchronization. You allow the task to initiate only when information about all the required functions has already been processed. Always remember that you cannot use concurrency in loops, as the next loop iteration often includes the use of data instances which are set up or worked on in the previous loop iteration.

19.9.2 High-Level Concurrency

The Java concurrency API provides various classes to perform the ideal parallelism in your programs. You can use the Lock or Thread classes, and you need to focus on the executors and fork/join facilities. This provides you access to concurrency at the highest level, like carrying out separate tasks with the use of multithreading. Here are the advantages of ensuring that concurrent tasks are specified at the highest level in your Java program:

1. The management of threads does not remain your responsibility when you allow Java to manage the intricate details of the required threads. The Flow API will manage these tasks and allow you to only create high-level functionality with a clean code.
2. Since this practice will ensure that the created threads are directly employed, they always occur in an optimized manner. The pool of threads is created according to the needs of the program once and then employed multiple times as and when required by your code. This allows you to automate several concurrency tasks.
3. The use of advanced features is possible with the use of thread groups and pools. The executors provide Future objects and ensure that you can use the main concept of concurrency for optimal benefits.
4. Your code is simple to execution in all JVM environments. The threading and other functions simply follow the available limits of the physical resources that they have available on different executing machines. This makes it easier to scale your operations and perform hardware migrations.
5. Your application is quicker to run, especially in environments that have access to the latest JVM and JRE versions. This is possible with continuous internal improvements and the ability to employ specific compiler optimizations.

19.9.2.1 Use of Immutable Objects

Another ideal practice is to avoid data race conditions in your concurrent programs. This is possible by creating immutable objects in Java. These objects are special because their attributes cannot be modified after their creation. This means that they cannot get altered during multiple executions that define the concurrency practice.

The String class in Java is an excellent example of this concept. It always generates a new object whenever mathematical operators are applied in the objects of this class. Immutable objects do not need to be synchronized since it is not possible to modify the important values in these classes. The modification of any object parameters simply gives rise to a new object, which means that the main attributes always remain fixed.

Another advantage of these objects is that you can always count on the consistency of the data that you have in your program, since class defining characteristics are always secure during program executions. However, if your program ends up creating too many objects, its performance can seriously go down because it will use greater memory and reduce the throughput. Complex objects that often contain other objects defined within them can create serious issues, and you must use this practice with care.

19.9.3 Controlling Sequential Executions

Sequential executions will always occur in a program. Your focus should always remain on reducing the amount of time you need to keep the other tasks locked up so the critical section of the program could perform its execution. Special attention is needed to ensure that you create locks and blocks that are time sensitive and ensure that your program does not suffer from an imbalance of serial and parallel tasks during execution.

This is a situation which is often simplified with the creation of high-level concurrency. This will allow you to create shorter critical sections and avoid situations where your library can quickly supply the required code during executions. A good example would be to use the available library documentation and make the best of the Java classes. One excellent code for this is the `compute()` method which is present in the ConcurrentHashMap class in Java.

You should also avoid placing blocking operations in your critical section. These are operations that can block the tasks that call them for use. These tasks are then only available after a particular instance is achieved, like reading a file or outputting data to the console. The performance of your program degrades because blocking operations are slow and may stop the rest of your program from operating until the results of these operations are produced.

19.9.4 Avoiding Lazy initialization

One trick for successful Java programming is to employ lazy initialization scheme. It is a method where you do not create an object before it is required for the first time in your program. This practice offers the advantage of reducing the load on the available heap for the Java program. You get on in your code, only creating objects that will be required in a specific execution.

However, the same situation can be a problem in concurrent programs, where you cannot control where an object may be required for the first time, especially with aggressive multithreading strategy. If you are using singleton classes, then it will not be possible to use singular objects. You can learn more about using the on-demand holder in Java by visiting the official guide on Oracle at https://community.oracle.com/docs/DOC-918906, which takes care of this problem and ensures that you can take advantage of lazy initialization as much as possible.

Summary

This chapter explains multiple themes that are related to the use of multithreading in the Java environment. We find that the use of multithreading brings in the application of parallel processing, which is often described in the form of concurrency in Java programming. There are several ways of implementing concurrency and designing code improvements that ensure that your Java program performs better than a typical sequential program.

We find that it is always the ideal practice to implement concurrency schemes at the highest level in your programs. In fact, with the availability of the worker thread pools and the functions offered by the executors, it is always better to leave the choice of selecting the threads and their concurrent behavior to the JVM itself. This will ensure that the ideal resources are employed each time the program operates, in various hardware environments.

We also share several important tips that help you avoid the common concurrency mistakes when programming in Java. We believe that if you focus on creating programs that use threads to match the available processors on the executing machines, it is possible to perform useful reactive programming. You can ensure that your program reads the available hardware resources before starting its working functions, and then make sure that it uses a pool of worker threads accordingly for maximum concurrency benefits.

There are times where you may never have to make difficult choices, especially with the ideal thread functionality offered by the Java Flow API. It certainly empowers concurrent programming and allows developers to focus on developing algorithms, which offer solutions to the client problems. The enhanced Java tools are sure to take care of the required thread, concurrency and reactive programming needs in each case in an ideal manner. Java is certainly an excellent choice for carrying out reactive programming, especially in various client/server applications.

In this chapter, we have learned the following concepts:

1. Reactive programming.
2. Multithreading and programming with multithreading.
3. Concurrency and advantages of concurrency
4. Concurrency API improvements.
5. Dealing with individual threads.

6. Synchronization blocks.
7. Understanding and resolving deadlocks.
8. Concurrent data structures.
9. Multithreading examples.
10. Designing concurrent Java programs.
11. Ideal tips for concurrent Java 11 programs.
12. High-level concurrency.
13. Controlling sequential executions.
14. Avoiding Lazy initialization.

In Chapter 20, we will explore the world of Spring and Hibernate. The chapter focuses on Spring Framework and Spring MVC. We will learn concepts like Inversion of Control and Dependency Injection. We will explore bean scopes like singleton and prototype. In addition, we will look into the role of controller and DispatcherServlet.

Multiple-Choice Questions

1. Which of the following is a type of multitasking?
   - **(a)** Thread based
   - **(b)** Process based
   - **(c)** Process and thread based
   - **(d)** None of the above
2. Which of the following is a thread priority in Java?
   - **(a)** Float
   - **(b)** Integer
   - **(c)** Double
   - **(d)** Long
3. Which of the following methods is utilized for launching a new thread?
   - **(a)** `run()`
   - **(b)** `launch()`
   - **(c)** `project()`
   - **(d)** `start()`
4. Which of the following is the default priority of a thread?
   - **(a)** 1
   - **(b)** 5
   - **(c)** 0
   - **(d)** 10
5. You can use the same thread object to launch multiple threads.
   - **(a)** True
   - **(b)** False

Review Questions

1. What is multithreading?
2. How is multithreading useful?
3. How will you write a concurrent program?
4. What is thread life cycle?
5. How can you make a thread wait for other threads?
6. What is deadlock? How do we avoid it?
7. What is reactive programming?
8. What are concurrent data structures?
9. What are the improved concurrent APIs?
10. What are the advantages of concurrent programs?
11. How will you set thread priority to give preference to one thread over other?

Exercises

1. Write a program that accepts stock stream data about current stocks and their prices. Write multithreading code to manage the stream and process in different threads.
2. Write a program for an ATM machine that dispenses money. Make sure this program uses multithreading reactive programming concepts. Also note that someone may transfer money from online backing or from a branch at the same time a user is withdrawing cash from the ATM, so you have to make sure the cash withdrawal process is secure in a way that a user should not be able to withdraw more than he/she has in his/her account.
3. Write a program using reactive programming for an online gambling application. Ensure that multiple people cannot bet on an event. You have to make sure everything happens in a timely manner.

Project Idea

Create an airline ticket booking application. Any person should be able to book a flight ticket from an airline's website, ticketing counter, or through a travel agent. Please note that flight prices are not fixed, they fluctuate based on demand. Hence, price may change between checking the price and booking a ticket. Also, tickets are limited per flight; thus, in a limited ticket availability case, the ticket may be booked by an agent while a user is trying to book it from the website. Hence, you have to make sure you use proper locks in case multiple threads are trying to book a ticket.

Recommended Readings

1. Javier Fernandez Gonzalez. 2017. *Mastering Concurrency Programming with Java 9*. Packt: Birmingham
2. Mayur Ramgir and Nick Samoylov. 2017. *Java 9 High Performance*. Packt: Birmingham
3. Paul Bakker and Sander Mak. 2017. *Java 9 Modularity: Patterns and Practices for Developing Maintainable Applications*. O'Reilly Media: Massachusetts